RocqStar: Leveraging Similarity-driven Retrieval and Agentic Systems for Rocq generation

Andrei Kozyrev[13*] **Nikita Khramov**[13*] **Gleb Solovev**[13] **Anton Podkopaev**[23]
[1]JetBrains Research, Germany [2]JetBrains Research, the Netherlands
[3]Constructor University Bremen, Germany [*]Contributed equally
{first}.{last}@jetbrains.com

Abstract

Interactive Theorem Proving was repeatedly shown to be fruitful combined with Generative Artificial Intelligence. This paper assesses multiple approaches to Rocq generation and illuminates potential avenues for improvement. We highlight the importance of thorough premise selection for generating Rocq proofs and propose a novel approach, leveraging retrieval via a self-attentive embedder model. The evaluation of the designed approach shows up to 28% relative increase of the generator's performance. We tackle the problem of writing Rocq proofs using a multi-stage agentic system, tailored for formal verification, and demonstrate its high effectiveness. We conduct an ablation study and demonstrate shows that incorporating multi-agent debate during the planning stage increases the proof success rate by 20% overall and nearly doubles it for complex theorems, while the reflection mechanism further enhances stability and consistency.

1 Introduction

In recent years, the advent of Generative Artificial Intelligence (AI) has accelerated the process of developing new software. However, there are studies [21] showing that users who use AI assistants tend to introduce more bugs and vulnerabilities into their code, compared to those who write code on their own. Formal software verification could help mitigate the issue of bugs and security flaws, as it ensures that the software operates correctly and reliably in compliance with the given specification. Under the assumption of a well-formed specification, formal verification provides strong guarantees and an acceptance criterion for the generated code. Interactive Theorem Prover (ITP) is a software tool that assists the user with the development of formal specifications and proofs. To date, there exist several ITPs, such as Rocq (former Coq) [1], Lean [5], Agda [13], Isabelle [19], and others. Rocq is a mature ITP, which has experienced more than 30 years of continuous development and improvement. Rocq has an extensive track record of high-impact projects. For example, Rocq was used to verify the correctness of the CompCert C compiler [15], the only C compiler, in which an extensive study found no bugs [32].

Verifying software has always been a rigorous process requiring much time and human effort. A number of solutions have been developed to help automate the process of theorem proving in Rocq. Proofs in Rocq are constructed from so-called *tactics*, which are elementary building blocks. Using tactics, the user manipulates the *proof state* — a data structure, which contains the current goal and the context of the proof. Thus, with every applied tactic, the task is transformed and could be solved recursively. Most solutions implement tactic-prediction approaches and employ beam search or a similar algorithm to navigate the search space. Tactician [3] is a KNN-based approach, which does similarity-based retrieval of tactics used in similar states. CoqGym [30] and Proverbot9001 [25] use Recurrent Neural Networks (RNNs), Graph2Tac [24] proposed a novel graph-based neural tactic

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: The 4th Deep Learning for Code Workshop.

prediction. Thakur et al. [26] and Kozyrev et al. [14] instead build generation pipelines around general-purpose, cloud-hosted LLMs, so that no heavy computations occur on the user's machine. CoqPilot [14], along with that, contributes a benchmarking framework and allows seamless integration of standalone tools into the workflow of Rocq's user.

Many approaches call attention to the importance of premise selection, *i.e.,* retrieving useful context information to advance generation. Yang et al. [31] introduced LeanDojo, a retrieval-augmented prover in Lean that significantly improves over non-retrieval baselines. Thompson et al. [27] present the Rango tool and report state-of-the-art performance on the CoqStoq benchmark, automatically synthesizing complete proofs for 32% of the theorems. The work highlights how strongly the well-formed context contributes to the success of Rango. Moreover, they show that *proof retrieval* is the most performant mechanism for premise selection. The proof retriever selects relevant previously completed proofs from the current project and provides them as references to the model. According to the evaluation, Rango proved 47% more theorems than the variant without a proof retriever. However, their retrieval mechanism assumes that two textually similar statements have proofs relevant to each other. In this work, we demonstrate that this assumption oversimplifies the relationship between statements and proofs and introduce a novel embedding model for Rocq statements. It is trained to predict the similarity between their proofs and achieves up to a 28% relative improvement on the evaluation set.

Another promising direction in generative theorem proving that we have identified is Agentic Systems. Research by Kozyrev et al. [14] shows that current Rocq generation methods mostly struggle with complex reasoning tasks. Algorithms that perform proof search on top of a tactic generator slow down dramatically and suffer performance degradation as theorem complexity grows, due to the properties of tree-based search. Other neural methods, which apply LLMs, suffer from the same problem due to the inability of the model to handle complex reasoning tasks [11]. Agentic systems are known to address these problems; however, to our knowledge, there were close to no attempts to build an autonomous agentic system for an ITP. We build an extensive Model Context Protocol (MCP) server for Rocq and implement an autonomous Agentic System over it, utilizing various problem-specific solutions, such as multi-agent debate. We conduct an evaluation and show that our agentic system strongly outperforms all other previously benchmarked solutions in the CoqPilot's work, raising the ratio of successfully proven theorems from 51% to 60%.

1.1 Contributions

The main contributions of this paper are the following.

RocqStar proof retriever We propose a novel approach for premise selection in Rocq. Rocq suffers from the data-scarcity problem that is common to most ITPs. Aggregating the largest publicly available repositories, one could expect to collect roughly 300 million tokens of Rocq, and about the same for Lean. In contrast, open-source Python corpora easily exceed 100 billion tokens. To tackle this issue we contribute a convenient standalone tool *BigRocq* to extract additional data from Rocq code, utilizing the nature of Rocq's system and the intermediate states of the proof. BigRocq bridges the gap between Automated Generation and Rocq's ecosystem. Using BigRocq, we mine a dataset of 76,524 statements with corresponding proofs from 4 big projects and train a self-attentive embedder model, which learns to predict how close the proofs of given statements will be. In addition, we provide a pipeline to reproduce such embeddings for an arbitrary project, which offers even better results. We integrate the solution as a new retrieval approach for selecting context theorems in CoqPilot and evaluate it using CoqPilot's benchmarking infrastructure. Compared to the baseline text similarity-based ranker, we show an improvement of 28% on the evaluation set. The BigRocq tool, the training dataset, and the code for training the embedder model are available at `https://github.com/JetBrains-Research/rocqstar-rag`. The embedder model's checkpoint is available at `https://huggingface.co/JetBrains-Research/rocq-language-theorem-embeddings`.

RocqStar agentic system Addressing the lack of research on applying agentic systems to ITPs, we build an autonomous system for generating Rocq proofs. A custom MCP server built on top of `coq-lsp` [6] handles the interaction with Rocq; its source code is available at `https://github.com/JetBrains-Research/rocqstar-agentic-system/tree/main/mcpServer`. Our approach follows a structured process consisting of *planning*, *execution*, and *reflection*. An ablation study shows that while naive planning has limited impact, effective planning based on the Multi-Agent Debate (MAD) framework plays a crucial role. Specifically, it yields a 20% relative improvement in

the overall proof success rate and nearly doubles the success rate on complex theorems with longer reference proofs (33% vs. 17%). Additionally, we demonstrate the benefits of the reflection mechanism, which improves the overall proof success rate from 48% to 66% and more than quadruples success on complex theorems; see § 4.3 for details. The evaluation results show that the RocqStar system solves up to 60% of theorems from the CoqPilot dataset. It is implemented using Koog[1], an innovative framework for building AI agents by JetBrains. The source code is available at `https://github.com/JetBrains-Research/rocqstar-agentic-system/tree/main/koogAgent`.

The remainder of the paper is organized as follows. § 2 describes our Similarity-Driven Retrieval mechanism. § 3 introduces the agentic system. § 4 presents an evaluation of the retrieval component (§ 4.1), the agent (§ 4.2), and an ablation study of the agentic system (§ 4.3). We describe the related work in § 5 and conclude in § 6.

2 Similarity-driven Retrieval

A known problem in Retrieval Augmented Generation (RAG), applied to the domain of Interactive Theorem Proving (ITP), is *premise selection* [28, 9]. Premise selection is the task of retrieving facts from a given knowledge base to help the model advance the proof. Huang et al. [8] and Xu et al. [29] highlight the importance of a well-formed context, showcasing that the presence of irrelevant context information degrades the model's performance.

We distinguish two ways of doing premise selection in Rocq. *Hint selection* — given a context C and a tactic with an unknown positional argument, e.g. `apply _`, the task is to yield potential candidates for the argument. *Proof selection*, in turn, given theorem statement S, focuses on choosing other statements with their respective proofs, so that their presence in the context of the generation request would help the model with the generation of the proof for statement S. Since the approach of applying general purpose models to proof generation is relatively new, most of the works [2, 12, 27, 31] on premise selection in Rocq and other ITPs focused on hint selection. However, Thompson et al. [27] and Kozyrev et al. [14] show that even a baseline proof selection significantly boosts the model's capabilities and is stronger than hint selection. The baseline proof selection presented in both works [27, 14], given the target statement s_* and a database of already proven theorems $[s_0, p_0], \ldots, [s_n, p_n]$, chooses theorems, statements of which have the maximum similarity to the target one. Similarity is defined by the BM-25 information retrieval technique [23] or the Jaccard similarity index. That results in packing the generator's context with theorems, based on how similar their statements are syntactically.

We propose a retrieval mechanism that improves the performance of the generator compared to the described baseline. During the generation of the target proof for the statement S we generally assume that the model benefits more from seeing similar proofs to the one it needs to generate, rather than from seeing similar statements with proofs dissimilar from the target one. Evaluation of our retriever in § 4.1 supports this supposition. One might assume that if statements s_* and s_i are similar, their respective proofs p_* and p_i are similar as well:

$$\text{similarity}(s_*, s_i) \implies \text{similarity}(p_*, p_i) \tag{1}$$

However, we show that this implication often **does not** hold. The heuristic of retrieving similar statements produces decent baseline results, but fails in complex cases, leaving room for improvement. We design our retrieval method to guide context selection based on the similarity of the proofs and show its practicality.

Let us define the proof similarity D_L as the Levenshtein edit distance computed over lists of tactics. Insertions and deletions of tactics have a unit cost, as in the standard Levenshtein formulation. The substitution cost between two tactics is proportional to the Levenshtein distance between their string representations. The resulting distance is normalized by the maximum proof length.

$$p_i = [tac_{i_0}, \ldots, tac_{i_m}], \quad l_i = |s_i|, \quad D_L(p_i, p_j) = \frac{\text{Lev}(p_i, p_j)}{\max(l_i, l_j)}$$

We conduct the following experiment to examine whether the relation in Equation (1) holds in practice. Considering 1,855,701 pairs of theorems from the IMM project[2], we compute correlations

[1] Koog `https://docs.koog.ai`

[2] IMM `https://github.com/weakmemory/imm`

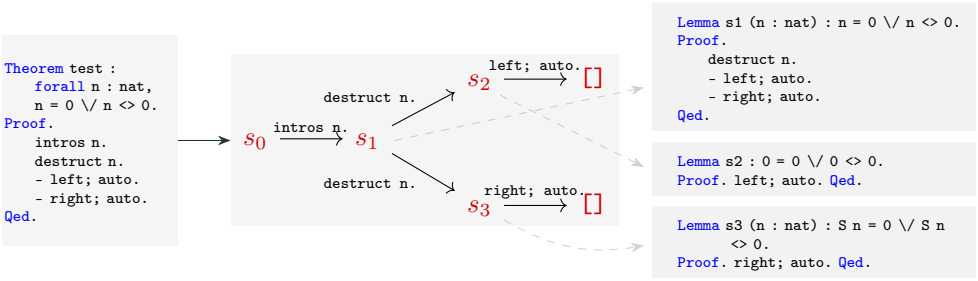

Figure 1: Processing theorems into trees. s_i denotes a state

between statement similarities and respective proof similarities. In summary, BM25-based statement similarity shows a weak negative relationship with the Levenshtein-based proof distance (Pearson r = –0.154, Spearman ρ = –0.171). The code to reproduce these experiments could be found in the *RocqStar-retriever* repository[3].

To assess the issue of ineffective proof selection, we try to find a function $f(s_i, s_j)$ that correlates with the defined proof-distance stronger than statement similarity does. In this work, we introduce a neural method that learns vector embeddings for Rocq theorem statements, training them so that the distance between any two vectors mirrors the similarity between the respective theorem' proofs.

2.1 Dataset mining

Along with other ITPs, Rocq struggles with data scarcity. To assess this issue, we mine additional data from the Rocq code. We utilize Rocq system's functionality, preprocess theorems, and transform sequential proof structures into trees. Fig. 1 illustrates this transformation process. Since every node in such a tree is a valid state, we can automatically construct a proof for it by recursively following its subtree edges. Extracting all intermediate statements together with their proofs allows us to expand any given Rocq theorem dataset by a factor roughly proportional to the average proof length. In our case, this resulted in an approximately fourfold increase in dataset size. The dataset format and further details are provided in Appendix B. We call the proposed tool *BigRocq* and make it publicly available as a standalone component of our system. The idea of mining additional training data from the intermediate states of the ITP is not new; Kogkalidis et al. [12] conducted analogous research for the Agda [13] language. Similar research for Rocq also takes place [30, 24]; however, some of those works are highly dependent on the deprecated ways of communication with Rocq's compiler [30] and do not support up-to-date versions of Rocq. In contrast, others implement similar ideas as a part of the training pipeline and do not allow for seamless reuse. Using BigRocq, we mine a total of 76,524 statements, collected from 344 files from 4 big Rocq projects.

2.2 Modeling

In our work, we formulate the problem as a self-supervised contrastive representation learning task and train a self-attentive embedder model [18]. Given a dataset $\mathcal{T} = \{(s_i, p_i)\}$, where s_i is a Rocq statement and p_i is its corresponding proof, and a similarity function $\mathrm{f}(\mathrm{proof}_i, \mathrm{proof}_j)$, defined between two proofs, we aim to learn a function

$$r : \mathcal{S} \times \mathcal{S} \to \mathbb{R},$$

which takes two statements $s_i, s_j \in \mathcal{S}$ as inputs and outputs a score approximating the similarity of their respective proofs p_i, p_j. In other words, $r(s_i, s_j) \approx f(p_i, p_j)$. This formulation allows the ranker r to assign scores to candidate statements relative to a target statement, thereby guiding retrieval towards those whose proofs are most likely to be useful.

In § 4, we evaluate the performance of the proposed model in the following task. Given a target statement s_* and a set of proven theorems $\mathcal{T}$, we aim to select k premises from $\mathcal{T}$ to be used as

[3]RocqStar retriever: `https://github.com/JetBrains-Research/rocqstar-rag/tree/main/experiments`

context for generating a proof of s_*. We take k most relevant theorems, according to the ranker r.

$$\mathrm{Top}_k(r, s_*) = \underset{(s_i,p_i)\in\mathcal{T}}{\arg\operatorname{top}_k} r(s_i, s_*)$$

We say that statement s_* is solved with the use of the ranker r if the generator g produces a valid proof for s_* given the premises selected by r.

$$\mathrm{Solve}(s_*, r, g) = \begin{cases} 1, & g\big(\mathrm{Top}_k(r, s_*),\ s_*\big) \text{ is a valid proof,} \\ 0, & \text{otherwise.} \end{cases}$$

Finally, the quality of the ranker is estimated by the number of theorems in the evaluation set that can be solved using r in combination with a given generator g.

One of the difficulties encountered during training is the U-shaped distribution of proof distances over random theorem pairs. Many short proofs were extremely similar to one another, while others were mostly too far apart, leaving a mid-range gap that hindered training. To mitigate this, we slightly modify the definition of $\mathrm{proof\_distance}(\cdot)$ introduced in § 2 incorporating an additional similarity term and noise for robustness.

$$\mathrm{proof\_distance}(p_i, p_j) = \alpha\, D_L(p_i, p_j) + (1-\alpha)\, D_J(p_i, p_j) + \gamma$$

$$D_J(p_i, p_j) = 1 - \frac{|p_i \cap p_j|}{|p_i \cup p_j|}$$

The coefficient $\alpha = 0.7$ was chosen heuristically based on the distribution plot and yielded the best performance in experiments. The noise γ is taken from $\mathcal{U}(-1e-3, +1e-3)$.

As we have already shown in § 2, statement similarity is a poor choice of r, as it shows a low correlation with the target function $\mathrm{proof\_distance}$. However, it still provides a strong baseline: in practice, similar theorems occasionally have similar proofs. Accordingly, our approach builds upon statement encoders, fine-tuning them to better align with the underlying proof structure. We fine-tune Microsoft's 108-million-parameter encoder CodeBert [7], originally pretrained on a combined corpus of programming and natural language texts. We also experimented with `gte-modernbert-base`[4] as the base model, but it did not yield notable improvements. As shown in § 4.1, the encoder without post-training performs on par with the Jaccard-similarity baseline, indicating that the unadapted model relies primarily on surface-level syntactic similarity rather than proof-related semantics. Our goal, therefore, is to adapt the encoder to capture this deeper semantic relation.

To achieve this, we train the model using the InfoNCE [20] loss. In our setting, the distribution of proof distances is imbalanced even after normalization. InfoNCE naturally handles this case by contrasting a limited number of positives against a set of negatives, ensuring that informative gradients are maintained. In particular, given a statement s, during dataset preprocessing we compute distances to other samples. We then mark a pair as positive if the distance between their proofs is less than a threshold τ_{pos}, and mark it as negative if it is greater than τ_{neg}. Given the hyperparameter k_{neg} and sets of positive and negative pairs P_s^+ and P_s^-, we compute a per-statement loss term $\mathcal{L}_s$ as follows:

$$\mathcal{L}_s = -\log \frac{\exp\big(\cos(z_s, z_p)/T\big)}{\exp\big(\cos(z_s, z_p)/T\big) + \sum_{j=1}^{k_{\text{neg}}} \exp\big(\cos(z_s, z_{n_j})/T\big)}$$

where $\cos$ is a cosine similarity between ℓ_2-normalized embeddings of statements, and $p \in P_s^+$, $n_j \in P_s^-$. On average, we observed smoother convergence for higher values of k_{neg}, which is consistent with findings by Chen et al. [4]. However, due to hardware limitations, we selected $k_{\text{neg}} = 100$ as a practical trade-off between convergence stability and computational cost.

Despite the adjustment of $\mathrm{proof\_distance}(\cdot)$, during training we experienced the problem of the model converging too quickly on *"easy"* negatives — pairs, whose proofs (and typically their statements) are already far apart in the raw distance space. To keep informative gradients flowing, we add *hard negative* pairs; with some probability we treat a pair of statements as negative if $\tau_{\text{hardneg}} \leqslant \mathrm{sim}(\mathrm{proof}_a, \mathrm{proof}_b) \leqslant \tau_{\text{neg}}$. Introduction of negative samples helped to stabilize the training process; we have observed a less steep training curve and better generalization overall. Other training hyperparameters are listed in Appendix C.

[4] gte-modernbert-base: `https://huggingface.co/Alibaba-NLP/gte-modernbert-base`

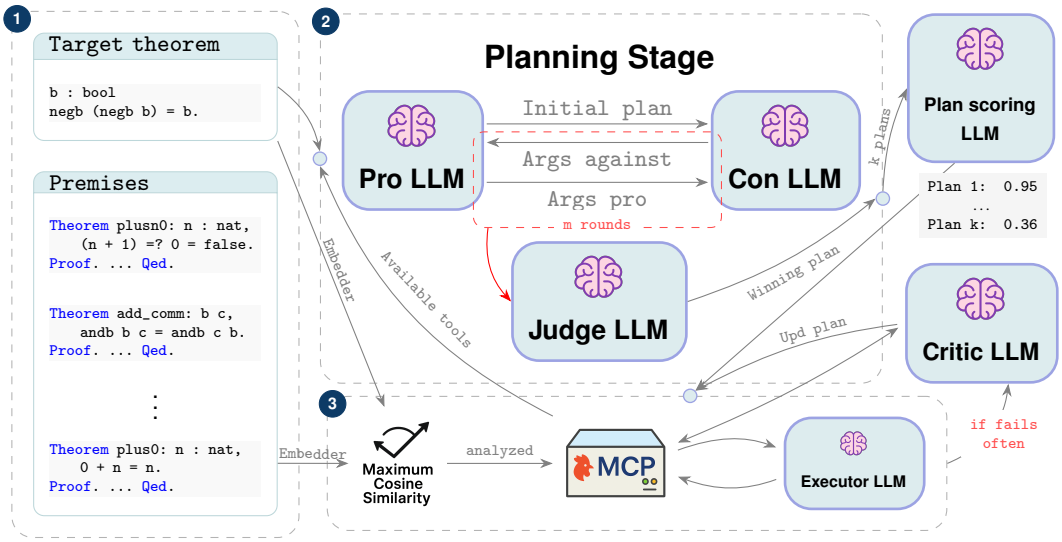

Figure 2: Agentic pipeline with RocqStar retriever.

3 Agentic System

Agent-based approaches are broadly used in code generation and repair tasks. Despite a large number of autonomous and semi-autonomous coding agents, they are not widely used in formal proofs generation and are not tailored to the Rocq specifics. To address this, we have implemented a RocqStar agentic system.

To allow interaction between the agent and Rocq's system, we develop a REST API server that provides a set of tools that are useful during the execution. We apply our domain knowledge and construct these tools to bring an agent-driven proving process as close as possible to a human-driven one. Examples of allowed function calls include checking validity of proofs, retrieving the valid prefix of given proof, gathering additional information about available entities in the context, and interacting with the context via performing commands like `Print ?a` to identify the type of an argument or `Search ?exp` to search for defined terms by a pattern. Toolset is described in detail in Appendix D. Interaction with Rocq's system is carried out through its language server, `coq-lsp` [6]. To conform with a commonly used Model Context Protocol (MCP) and allow seamless agent interaction with the environment through tools, we implement an MCP server that wraps the REST API server. Among the provided tools, the most important is the proof-checking tool. It not only verifies whether a proof is valid but, in case of an error, returns detailed diagnostic information: the error message, its exact location, the valid prefix preceding the error, and the remaining goals after that prefix. This functionality allows the agent to maintain awareness of the current proof state and leverage partial proof progress.

3.1 Agent Logic

The input to the agent is presented as a target theorem without a proof and a file where it was declared, see box 1 of Fig. 2. Agent's pipeline is logically split into two main stages: *planning* and *execution*. In the planning phase, multiple language models rigorously work out the strategy for the further implementation. During execution agents follow the plan aiming to generate the correct proof.

Planning Stage We employ the idea of multi-agent debates to generate a strategy for proving the given theorem. Specifically, two LLMs engage in a discussion: one proposes and defends an initial plan (*pro* LLM), while the other critiques it (*con* LLM); see box 2 in Fig. 2. After several debate rounds, the entire message history is passed to a *judge* LLM, which determines the winner and produces the final plan. Using this procedure, we generate k candidate strategies. These are then evaluated by a *plan scoring* LLM that assigns each a numerical score (the higher, the better). Finally, the top-l plans are selected and forwarded to the *Execution Stage*; see box 3 of Fig. 2.

Execution Stage For each of the selected plans, we run an *executor* agent that follows it step by step, invoking tools from the provided tool set — proof checker, context–inspection queries, search commands, and others, as atomic actions. Through these tool calls, the agent interacts with the environment via the MCP server. In addition to this iterative execution, we employ a reflection mechanism that monitors the progress of the proof and adjusts the strategy when necessary. We track how many consecutive erroneous proof attempts occur, and once this number exceeds a predefined threshold (set to five during evaluation), a *critic* model is called to assess the current proof state and identify deviations from the intended strategy. After that, we retrieve theorems along with their proofs, whose top-level goals are similar to the currently remaining goal, according to the cosine similarity between their RocqStar-ranker embeddings. We prompt the LLM to explain which tactic sequences could be helpful to finish our proof. We gather the generated criticism and send it to the *replanner* LLM to refine the current plan along with similar proofs and their analysis. The replanner is a separate language model that revises the plan based on the critic's feedback and the retrieved examples. The whole message history is sent back to the *executor* agent. During the execution of each plan, n tool calls are allowed. If valid proof is not found after n tool calls, we denote the plan as failed. In this case, we ask a *plan failure summarizer* LLM to generate a short explanation of why the strategy execution failed and what happened during it. Then this summarized explanation is sent to the new execution stage with the next selected plan. This procedure is repeated until the correct proof is found or there are no more strategies to execute.

4 Evaluation

To evaluate our approach, partially and as a whole, we use the CoqPilot benchmarking framework. We required a dataset with a large number of human-written theorems and proofs. To compare our solution to existing ones, we decided to re-use the dataset by Kozyrev et al. [14]. It is limited to 300 theorems from the IMM project [22], which was suitable for us in terms of computational and financial costs. The theorems are partitioned into three groups, corresponding to the difficulty level. The length (in tactics) of the human-written reference proof of the theorem estimates its difficulty. The sizes of each group are chosen with respect to the initial distribution of proof lengths in the project. Final group sizes and length ranges of each group could be found in Table 2. From now on, we will refer to the described dataset as the *IMM-300* dataset. For smaller ablation studies we additionally prepared *IMM-50*, a 50-theorem subset of IMM, constructed with the same procedure. No theorems from the dataset were present in the training set of the RocqStar ranker embedding model. Moreover, the training set only contained partial theorem goals, no initial statements. Split of both datasets into groups, details, and limitations are described in Appendix A. Computational and financial resources used for experiments are described in Appendix F.

4.1 Retrieval Mechanism

We integrate our retrieval mechanism as a ranker into CoqPilot and evaluate it on the IMM-300 dataset with different models under the hood. To assess its performance, we compare our approach against two baselines: (i) an untrained embedder model (we use `gte-modernbert-base`, with `codebert-base` yielding comparable results), and (ii) a lexical similarity baseline based on the Jaccard index. In the latter, given a target theorem statement s_* and a set of proven theorems $[s_0, p_0], \ldots, [s_n, p_n]$, it ranks the theorems in descending order of $J(s_*, s_i)$, where $J(s_*, s_i)$ is the Jaccard-similarity index and S_{s_i} is a set of tokens inside a statement. The statement is split into tokens by whitespaces, commas, etc. Jaccard-similarity index is semantically almost the same as the BM-25 metric and produces the same numerical results. For each theorem in the dataset, we take theorems within the same file, sort them using the ranker (Jaccard, ModernBert, or RocqStar, respectively), take the k most relevant ones (k is equal to 7 in our experiments) and send a request to the model to generate the completion. The chosen theorems are being sent as a few-shot prompt. Generation for each theorem is requested 12 times. If the Rocq's system accepts any of the proofs, the theorem is considered solved. The target metric in our evaluation is the ratio of solved theorems. The evaluation results are presented in Table 1. The reported values denote mean success rates, and the $\pm$ intervals correspond to the standard deviation across three independent runs.

As can be seen from Table 1, our RocqStar ranker consistently outperforms both the Jaccard baseline and the untrained ModernBert encoder, demonstrating reliable gains across all evaluation groups. Most of the performance increase could be seen in the second group; we interpret these results as

Group	$\leqslant 4$			$5-8$			$9-20$		
Ranker	Jaccard	ModernBert	**RocqStar**	Jaccard	ModernBert	**RocqStar**	Jaccard	ModernBert	**RocqStar**
GPT-4o	48% ± 5%	44% ± 4%	**51%** ± 5%	18% ± 4%	21% ± 5%	**25%** ± 3%	11% ± 4%	8% ± 4%	**14%** ± 5%
Claude 3.5	58% ± 5%	57% ± 3%	**61%** ± 4%	28% ± 5%	30% ± 3%	**36%** ± 5%	16% ± 5%	16% ± 5%	**21%** ± 5%

Table 1: Model performance under different ablations across all evaluation sets.

Reference proof length	$\leqslant 4$	5 – 8	9 – 20	Total
Group size	131	98	71	300
OpenAI GPT-4o	50%	26%	15%	34%
OpenAI o1	66%	31%	8%	41%
Deepseek R1	58%	29%	11%	37%
Claude 3.5 Sonnet	73%	41%	27%	51%
LleMMa 7B	24%	11%	1%	15%
Tactician (`synth`)	45%	23%	10%	29%
Rango	38%	18%	8%	25%
RocqStar Agent	**76%**	**56%**	**38%**	**60%**

Table 2: Measuring the performance of different Rocq generation methods via CoqPilot

follows. For short theorems in the first group, the assumption that similar statements imply similar proofs often holds; therefore, all rankers perform comparably. For complex theorems from the third group, it rarely happens that two theorems have significantly similar proofs, resulting in less advancement space for the model.

4.2 Agentic System

We evaluate our agentic system on the IMM-300 dataset, pursuing the goal to solve as many theorems as possible. For all of the parts of the planning stage, we use the Claude 3.5 Sonnet model, performing two rounds of debates between actors. Four plans are generated, and two are chosen for further execution. During execution, 20 tool calls are allowed from the MCP server. Additionally, after five proof-checking calls, the critic model (Claude 3.7 Sonnet) is invoked and analyzes whether a deviation from the initial plan has occurred. We use Claude 3.5 Sonnet for the execution and re-planning, and Google Gemini Flash 2.0 for other tasks, due to the necessity of a big context. Results of the evaluation are shown in Table 2.

As shown in Table 2, our agentic system outperforms other benchmarked models inside the CoqPilot framework. The strongest model so far was Claude 3.5 Sonnet, which achieves 51% accuracy on the dataset, given 12 retries for each theorem. RocqStar agent achieves 60%, showing vigorous improvement. In terms of financial costs, we estimate a run of an agent on one theorem at 1.3 US dollars, compared to 0.25 US dollars for 12 requests to the pure Claude 3.5 Sonnet in CoqPilot. Along with five language models invocated through the CoqPilot framework, we have compared our solution to other Rocq generation approaches, such as Tactician and Rango. On our IMM-300 dataset both solutions showed a result comparable to CoqPilot with OpenAI GPT-4o as the generator model.

4.3 Ablation study

We conduct an ablation study to analyze the contribution of individual components of the agentic pipeline to the overall success rate. In particular, we investigate the effects of removing (1) the *Multi-Agent Debate (MAD)* layer responsible for iterative plan refinement, (2) the *Planning* stage entirely, (3) the *RocqStar retrieval* module, and (4) the *Reflection* mechanism responsible for forced retrieval, criticism, and replanning. All experiments are performed on the IMM-50 dataset, with all other system components kept unchanged. The results are summarized in Table 3. **Planning** Considering that software-verification tasks cannot be solved ad hoc, without explicit planning, we measure how removing the MAD layer and reverting to single-pass planning affects the proportion of successfully proved theorems. We run two versions of the agent: one generates plans via MAD,

Reference proof length	$\leqslant 4$	5 – 8	9 – 20	Total
Group size	22	16	12	50
Agent	**91%**	**56%**	**33%**	**66%**
Agent w/o MAD	86%	44%	17%	56%
Agent w/o Planning	86%	50%	17%	58%
Agent w/o RocqStar retrieval	86%	50%	33%	62%
Agent w/o Reflection	73%	44%	8%	48%
Claude 3.5 Sonnet	86%	37%	8%	52%

Table 3: Ablation study of Multi-Agent Debate at planning stage

and the other produces a single plan in one LLM call without further refinement. Additionally, we include a configuration with the *Planning* stage entirely disabled, where the executor immediately attempts to construct a proof without any plan. This comparison shows that an agent without planning performs nearly identically, or slightly worse, than one guided by a poor single-pass plan, suggesting that a suboptimal plan does not provide advantage. In contrast, multi-step planning via MAD yields a consistent improvement across all groups, confirming the importance of structured plan refinement. An example of how MAD repairs a previously unsuccessful plan is presented in Appendix E.

RocqStar retrieval To further justify the improvements of the agentic pipeline, we evaluate the agent with its retrieval component replaced by the baseline Jaccard index. The observed drop in performance compared to the full pipeline highlights the importance of the RocqStar retrieval approach, while the gain over the Claude 3.5 Sonnet model demonstrates the effectiveness of the proposed agentic design.

Reflection Finally, we disable the reflection mechanism that triggers forced retrieval and replanning after failed attempts. Without reflection, the agent loses its ability to recover from early mistakes, leading to a noticeable degradation of performance, especially on longer and more complex proofs. The result shows that reflection complements planning by enabling recovery from failed reasoning trajectories.

5 Related Work

Many Rocq generation methods improve generation using Retrieval Augmentation. Most of those works solve the hint selection problem [2, 27], described in § 2. Those approaches build proofs tactic by tactic, retrieving relevant lemmas or definitions to use in the next step. The problem of searching for existing proofs that could advance the generation is barely described in the literature. CoqPilot [14] and Rango [27] pack the context for the generator model with theorems most similar to the one we are solving. Our work proposes a novel method of doing premise selection and shows improvement over the baseline from previous works [14, 27].

In our Multi-Agentic system, we distribute responsibility over different agents. Differentiating between models that handle natural reasoning and those that handle coding is common practice in agentic systems. The work of Li et al. [16] proposes a similar task force split into Thinker, Solver, Critic, and Debug agents. Liang et al. [17] introduces a Multi-Agent debate framework, shows that this approach encourages divergent thinking, and demonstrates its usability in complex reasoning tasks. We show that planning is essential for the formal verification pipeline. Theorem-proving demands a clear, high-level picture of the proof before executing any code. Running a multi-agent debate at the planning stage ensures rigorous evaluation of different approaches before interacting with Rocq's system. We produce several plans for further execution. In a manner, close to Islam et al. [10], we assign scores to plans and run them in order of score decrease. To our knowledge, there were close to no attempts to building Agentic systems for ITPs. Yang et al. [31] have shown an initial proof of concept of an agent for Lean; however, their agent lacks automaticity, the pipeline incorporates only minimal tooling, and does not possess an explicit planning stage.

As a user interface, we utilize CoqPilot to integrate into the common Rocq's programmer pipeline. CoqPilot is a VSCode[5] plugin, facilitating access to Rocq generation methods for end-users.

[5]VSCode: `https://code.visualstudio.com`

6 Conclusion

We have presented a method to enhance retrieval-augmented generation in Rocq via leveraging neural premise selection using a self-attentive embedder model. We evaluated our proposed solution on a dataset of 300 Rocq theorems with two different generator models under the hood and showed a noticeable improvement of up to 28% relative to the baseline. Our result suggests that proof-aware premise selection considerably improves generation quality, particularly for medium-difficulty theorems, where the gap between statement similarity and proof similarity becomes more significant.

Our work pioneers the use of Agentic Systems applied to Formal Verification. We have implemented an advanced pipeline that includes rigorous planning via multi-agent debate, domain-specific tooling, and an adaptive executor–critic loop that iteratively refines proofs based on partial progress. We conclude that our RocqStar agent shows promising results, surpassing strong baselines and highlighting the applicability of agentic systems in the domain of theorem proving. The ablation study further demonstrates that both multi-agent planning and reflection are key to maintaining stable reasoning and achieving consistent improvements in the proof's success rate.

Acknowledgments and Disclosure of Funding

We thank Ekaterina Verbitskaia, Ivan Kabashnyi, Maksim Rozenberg, and Pavel Guliaev for their valuable feedback on this work. We are also grateful to the JetBrains Koog team for providing a hands-on and well-designed framework for building AI agents. Finally, we thank the Dynamic Program Analysis Team (Egor Klimov, Nikita Dukin, and Saga Rut Sunnevudóttir) for their help in analyzing the agent's execution traces.

References

[1] Yves Bertot and Pierre Castéran. 2013. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media. https://doi.org/10.1007/978-3-662-07964-5

[2] Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. 2020. Tactic learning and proving for the Coq proof assistant. *arXiv preprint arXiv:2003.09140* (2020). https://doi.org/10.29007/wg1q

[3] Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. 2020. The tactician: A seamless, interactive tactic learner and prover for coq. In *International Conference on Intelligent Computer Mathematics*. Springer, 271–277. https://doi.org/10.1007/978-3-030-53518-6_17

[4] Changyou Chen, Jianyi Zhang, Yi Xu, Liqun Chen, Jiali Duan, Yiran Chen, Son Tran, Belinda Zeng, and Trishul Chilimbi. 2022. Why do we need large batchsizes in contrastive learning? a gradient-bias perspective. *Advances in Neural Information Processing Systems* 35 (2022), 33860–33875.

[5] Leonardo De Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 2015. The Lean theorem prover (system description). In *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*. Springer, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

[6] Emilio Jesús Gallego Arias et al. 2022. Visual Studio Code Extension and Language Server Protocol for Coq. https://github.com/ejgallego/coq-lsp

[7] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:2002.08155 [cs.CL]

[8] Yue Huang, Yanbo Wang, Zixiang Xu, Chujie Gao, Siyuan Wu, Jiayi Ye, Xiuying Chen, Pin-Yu Chen, and Xiangliang Zhang. 2025. Breaking Focus: Contextual Distraction Curse in Large Language Models. *ArXiv* abs/2502.01609 (2025). https://api.semanticscholar.org/CorpusID:276107466

[9] Geoffrey Irving, Christian Szegedy, Alexander A Alemi, Niklas Eén, François Chollet, and Josef Urban. 2016. Deepmath-deep sequence models for premise selection. *Advances in neural information processing systems* 29 (2016).

[10] Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. Mapcoder: Multi-agent code generation for competitive problem solving. *arXiv preprint arXiv:2405.11403* (2024).

[11] Bowen Jiang, Yangxinyu Xie, Zhuoqun Hao, Xiaomeng Wang, Tanwi Mallick, Weijie J Su, Camillo J Taylor, and Dan Roth. 2024. A peek into token bias: Large language models are not yet genuine reasoners. *arXiv preprint arXiv:2406.11050* (2024).

[12] Konstantinos Kogkalidis, Orestis Melkonian, and Jean-Philippe Bernardy. 2024. Learning structure-aware representations of dependent types. *Advances in Neural Information Processing Systems* 37 (2024), 65095–65118.

[13] Wen Kokke, Jeremy G. Siek, and Philip Wadler. 2020. Programming language foundations in Agda. *Science of Computer Programming* 194 (2020), 102440. `https://doi.org/10.1016/j.scico.2020.102440`

[14] Andrei Kozyrev, Gleb Solovev, Nikita Khramov, and Anton Podkopaev. 2024. CoqPilot, a plugin for LLM-based generation of proofs. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 2382–2385. `https://doi.org/10.1145/3691620.3695357`

[15] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*.

[16] Jierui Li, Hung Le, Yingbo Zhou, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. 2024. Codetree: Agent-guided tree search for code generation with large language models. *arXiv preprint arXiv:2411.04329* (2024).

[17] Tian Liang, Zhiwei He, Wenxiang Jiao, Xing Wang, Yan Wang, Rui Wang, Yujiu Yang, Shuming Shi, and Zhaopeng Tu. 2023. Encouraging divergent thinking in large language models through multi-agent debate. *arXiv preprint arXiv:2305.19118* (2023).

[18] Zhouhan Lin, Minwei Feng, Cicero Nogueira dos Santos, Mo Yu, Bing Xiang, Bowen Zhou, and Yoshua Bengio. 2017. A structured self-attentive sentence embedding. *arXiv preprint arXiv:1703.03130* (2017).

[19] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer. `https://doi.org/10.1007/3-540-45949-9_5`

[20] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. 2018. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748* (2018).

[21] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2022. Do users write more insecure code with ai assistants?(2022). *arXiv preprint arXiv:2211.03622* (2022).

[22] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the gap between programming languages and hardware weak memory models. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–31.

[23] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.

[24] Jason Rute, Miroslav Olšák, Lasse Blaauwbroek, Fidel Ivan Schaposnik Massolo, Jelle Piepenbrock, and Vasily Pestun. 2024. Graph2Tac: Learning Hierarchical Representations of Math Concepts in Theorem proving. *arXiv preprint arXiv:2401.02949* (2024). `https://doi.org/10.48550/arXiv.2401.02949`

[25] Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner. 2020. Generating correctness proofs with neural networks. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 1–10. https://doi.org/10.1145/3394450.3397466

[26] Amitayush Thakur, Yeming Wen, and Swarat Chaudhuri. 2023. A language-agent approach to formal theorem-proving. *arXiv preprint arXiv:2310.04353* (2023). https://doi.org/10.48550/arXiv.2310.04353

[27] Kyle Thompson, Nuno Saavedra, Pedro Carrott, Kevin Fisher, Alex Sanchez-Stern, Yuriy Brun, João F Ferreira, Sorin Lerner, and Emily First. 2024. Rango: Adaptive Retrieval-Augmented Proving for Automated Software Verification. *arXiv preprint arXiv:2412.14063* (2024).

[28] Josef Urban. 2004. MPTP–motivation, implementation, first experiments. *Journal of Automated Reasoning* 33 (2004), 319–339.

[29] Xiaohan Xu, Chongyang Tao, Tao Shen, Can Xu, Hongbo Xu, Guodong Long, Jian guang Lou, and Shuai Ma. 2024. Re-Reading Improves Reasoning in Large Language Models. arXiv:2309.06275 [cs.CL] https://arxiv.org/abs/2309.06275

[30] Kaiyu Yang and Jia Deng. 2019. Learning to prove theorems via interacting with proof assistants. In *International Conference on Machine Learning*. PMLR, 6984–6994. https://doi.org/10.48550/arXiv.1905.09381

[31] Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. 2023. Leandojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems* 36 (2023), 21573–21612.

[32] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294. https://doi.org/10.1145/1993498.1993532

A IMM Evaluation Dataset

The collected *IMM-300* dataset from the CoqPilot [14] work includes only theorems with proofs of length no more than 20. For that reason the bucket with the most difficult theorems is labeled 9 — 20 tactics. This decision has been made, reflecting CoqPilot's original focus on subgoals and shorter lemmas. Theorems of length no more than 20 tactics account for 83% of all proofs in the IMM project. As we take the same dataset, it possesses the same limitations. Therefore, we have not evaluated our solution on theorems, for which the reference proof contains more than 20 tactics. However, such theorems are quite rare.

The exact list of theorems used in each group could be found in the repository of the CoqPilot project: `https://github.com/JetBrains-Research/coqpilot/blob/main/etc/docs/benchmark`.

A common problem with testing pipelines that include general-purpose LLM providers, such as OpenAI, is data contamination. We are aware, that the model could have possibly seen the human-written proofs, as the IMM project has been publicly available since a while. However, firstly, the model sees neither the theorem name, for which it is generating the proof, nor the proof goal exactly as it was in the initial file. As we treat them as proof states, rather than theorems, an LLM receives it in an equivalent, but slightly modified way. Secondly, as many of our experiments have shown, various quality of premise selection drasticly changes the behavior of the model. That hints that the model is not able to memorize all theorems and proofs. Lastly, data contamination issue was one of the things we had in mind, while developing BigRocq. One could pass a Rocq project into BigRocq as input, and for each theorem retrieve the sub-state, that is achieved after k steps. On an example of $k = 2$, the following theorem:

```
Lemma eq_trans (A : Type) : forall (x y z : A), x = y -> y = z -> x = z.
Proof.
    intros x y z Hxy Hyz.
    rewrite Hxy. (* State: (A : Type) (x y z: A) (Hxy: x = y) (Hyz: y = z) : y = z *)
    rewrite Hyz.
    reflexivity.
Qed.
```

Could be automatically tranformed into the following one:

```
Lemma eq_trans_modified (A : Type) (x y z: A) (Hxy: x = y) (Hyz: y = z) : y = z.
Proof.
    rewrite Hyz.
    reflexivity.
Qed.
```

The higher k is chosen, the smaller would be the chances of data leakage, as the produced sub-state gets further and further from the original theorem.

B Encoder Training Dataset

One of the limitations of our BigRocq tool is that it cannot process theorems that contain so-called *goal selectors*. The following example illustrates how they work.

```
Theorem test2nat1 : forall n : nat, n = 0 \/ n <> 0.
Proof.
    destruct n.
    - left; auto.
    - right; auto.
Qed.
```

This example could be rewritten with the use of goal selectors to the following proof:

```
Theorem test2nat1 : forall n : nat, n = 0 \/ n <> 0.
Proof.
    intros n.
    destruct n.
    all: try (left; auto) || (right; auto).
Qed.
```

Due to the limited information we get from the Coq-LSP, our heuristic algorithm of transforming the proof into a tree breaks down. We cannot augment such theorems. The authors of CoqGym [30] also

Parameter	Value
algorithm	AdamW (0.9, 0.99, e−2)
schedule	linear warmup (10)
lr	4e−6
batch size (stmts)	16
dropout	0.1

(a) Optimization hyperparameters

Parameter	Value
embedding dim	768
max sequence length	128
(positive, negative) threshold	(0.3, 0.65)
threshold hard neg.	0.45
hard negatives prob.	30%

(b) Model&Dataset hyperparameters

Table 4: Hyperparameters of the embedder training

explicitly state that they do not handle theorems with goal selectors. They state that in their dataset, goal selectors occur in less than 1% data. The situation has changed since the work was published; the feature is now used more often but is still relatively rare. Goal selectors are an issue to be solved, and we are working on a solution by extracting some additional information from Rocq's system through Coq-LSP.

The dataset is stored as a collection of JSON files and, due to its relatively small size, is stored within the repository, in the sub-directory with the model training code: `https://github.com/JetBrains-Research/rocqstar-rag/tree/main/proof-embeddings/data`.

Dataset is split into training, validation, and test sets with proportions of 70%, 20%, and 10% respectively. Theorems from the same file do not appear in different sets. Parameters of building the dataset are listed in Table 4b. Pair of statements is considered as negative, if the `proof_distance` between them is greater than 0.65, and positive if it is less than 0.3. If the distance is in range $[0.45, 0.65]$, with probability of 30% it is also considered to be a negative pair (see *hard negatives* in § 2.2).

C Encoder Details

The hyperparameters used for training the embedder model are listed in Table 4. We have used `microsoft/codebert-base` as the base model and trained our embedder for 22000 steps with a batch size 16, since we use many negative samples in the loss. We applied a dropout of 0.1 on the last layer of the model; the embedding dimension is 768, and the maximum sequence length is 128. We use AdamW optimizer with a linear warmup schedule for 10% of the training steps.

C.1 Visualizing RocqStar vs. Baseline Premise Selection

Here we try to illustrate the difference between different rankers and show an example of a theorem from IMM project, where our ranker outperforms the baseline. Figure 3 presents such an example.

```
Lemma ext_sb_trans :  transitive ext_sb.
Proof using.
  unfold ext_sb; red; ins.
  destruct x,y,z; ins; desf; splits; eauto.
  by rewrite H2.
Qed.
```

```
Lemma ext_sb_irr :  irreflexive ext_sb.
Proof using.
  unfold ext_sb; red; ins.
  destruct x; ins; desf; splits; firstorder.
  lia.
Qed.
```

Figure 3: Theorems with dissimilar statements and similar proofs

If we measure the distance between theorems from Figure 3 using the conventional Jaccard distance, which is used by default in CoqPilot, we get 0.67:

$$\begin{aligned}\text{Jaccard\_distance}(t1, t2) &= 1 - \frac{|\{\texttt{transitive}, \texttt{ext\_sb}\} \cap \{\texttt{irreflexive}, \texttt{ext\_sb}\}|}{|\{\texttt{transitive}, \texttt{ext\_sb}\} \cup \{\texttt{irreflexive}, \texttt{ext\_sb}\}|} \\ &= 1 - \frac{1}{3} = 0.67\end{aligned}$$

Jaccard ranker focuses only on statement similarity, which in this case is relatively small, the only similar parts are highlighted with red. Jaccard would probably not select theorem `ext_sb_irr` as a premise for theorem `ext_sb_trans`; however, they have similar proofs and one could help the model to generate the proof for the other. Similar parts of the proofs are highlighted with yellow. If

we measure the distance between these theorems using the `proof_similarity` metric we define, we get 0.32, and our trained model yields 0.28. When using our ranker, it is probable that one theorem would be selected as a premise for the other.

$$\text{proof\_sim}(t1, t2) = 0.32$$
$$\text{embedder\_pred}(t1, t2) = 0.28$$

D Agent toolset

Below are the tools that the agent uses to interact with Rocq's system. The session is a utility abstraction, mainly handled by our middleware server under the MCP. It manages sessions and creates a new one when the agent starts proving a new theorem. Sessions are introduced to speed up type-checking and reduce overhead. When the session is started, we create a file, copy all required theorem's context into it, type-check the context using Coq-LSP, and then start executing commands and continuously checking generated proofs in the context of this session.

- `list_coq_files`: Returns a list of all Coq files in the project.
- `get_theorem_names`: Retrieves theorem names available in the file, including the target theorem.
- `get_theorem_names_excl`: Retrieves the available theorem names from a file with the target theorem excluded from the list.
- `get_current_target_state`: Returns the proof stage for the target theorem in the current session.
- `get_theorem_with_proof`: Given the theorem's name, returns the theorem with its proof.
- `check_proof`: Validates a proof (or a part of a proof) in the context of a session and returns either of the following:
  (i) That there are no more goals to prove
  (ii) Provided proof produces no errors, but the goal is not fully solved. Returns: updated goal state
  (iii) The current goal is solved, but there are more goals at other depth levels. Returns: first unsolved goal at the closest depth level
  (iv) Provided proof produces errors. Returns: error message
- `get_similar_proofs`: Given theorem goal/statement as a string, it uses RocqStar ranker to retrieve theorems that are similar to the input statement and returns 15 most similar ones.
- `about_term`: Uses `About` Rocq's Command in the current session. Accepts the term name as an argument. Outputs the term's definition and a short description from the Coq-LSP.
- `search_pattern`: Uses Rocq's `Search ?exp` to search for a pattern in the current session's file. An example of a valid command: `Search (?a + ?b = ?b + ?a)`. It could be useful for finding lemmas that could be used in the proof.
- `print_term`: Prints a term in the current session file. Uses Rocq's `Print` command. Accepts the term name as an argument. Outputs the term's definition.
- `check_term`: Checks a term in the current session's file. Uses Rocq's `Check` command. Similarly to `print_term` or `about_term`, but outputs only the type of the term. In the case of a theorem, it outputs its statement.

E Plan refinement via Multi-Agent Debate

We will illustrate the difference between the two plans; one was generated using a single call to a model, and the other was generated using multi-agent debate. The theorem we are trying to prove is the `sub_IPROP` from the `src/imm/SubExecution.v` file, that states:

```
Lemma sub_IPROP : subseteq (IPROP G') (IPROP G).
```

The following plan was produced on the planning stage via a single request to the model, without multi-agent debate:

Let me outline a proof strategy for showing that IPROP G' is a subset of IPROP G.
First, I want to understand the context better. The lemma involves two grammars G and G', and we need to show that the improper symbols of G' are contained within the improper symbols of G. Let me check the definitions and related theorems.

print_term: "IPROP"
Understanding the definition of IPROP will help us formulate the proof. This looks like a lemma about grammar properties, where we need to show that if a symbol is improper in G', it must also be improper in G.
My proposed strategy:

1. Use the definition of subset ($\subseteq$) to unfold the goal.
2. For any symbol X in IPROP G', show it's also in IPROP G.
3. This will likely require using the definitions of improper symbols and their relationship between grammars G and G'.

I expect we'll need to:

- Start with intros X H to get a symbol X and hypothesis that it's in IPROP G'.
- Use the definition of improper symbols to understand why X is improper in G'.
- Show that these same conditions must hold in G.

Before proceeding with the actual proof steps, I'd like to see if there are any similar theorems or relevant properties about IPROP that might help us.

search_pattern: "IPROP"

The proposed proof strategy is quite vague, too wordy, and imprecise. With this plan, the executor agent started by fetching similar theorems and then got stuck in a cycle of ad-hoc attempts to check incorrect proofs. In contrast, the winning plan was precise, concise and clear. It had a top-level structure of the proof incorporated into the plan and made a winning solution to use the `search` command, which helped to finish the proof:

```
Proof.
    unfold IPROP.
    rewrite sub_W.
    rewrite sub_is_ta_propagate_to_G.
    done.
Qed.
```

Below, as a comparisson, is the plan, yielded by the multi-agent debate:

The winning proof plan follows these steps:

1. Initial Analysis:
   - Use rewrite /IPROP /subseteq to fully understand the expanded goal
   - Check G and G' relationship via about_term and search_pattern
   - Look for helper lemmas about IPROP transformations
2. Proof Structure:

```
Proof.
rewrite /IPROP /subseteq.
(* See expanded goal structure *)
intros x Hx.
(* Analyze Hx's type carefully *)
(* Use discovered helper lemmas *)
```

3. Reasoning Steps:
   - Work with explicit set inclusion based on subseteq definition
   - Use any discovered lemmas about G/G' relationship
   - Preserve flexibility to adapt based on actual definitions

B's plan wins because it:

- Prioritizes understanding definitions before tactical decisions
- Makes systematic use of available tools
- Avoids assumptions about existential structure
- Maintains flexibility while having concrete next steps

F Experiments compute resources

Our experiments in the evaluation part mainly use cloud LLM providers and therefore require minimum compute, but comprehensive financial resources.

Embedder training During our final training run of the embedding model for RocqStar ranker, the model consumed roughly 43 GB of GPU-process memory and only about 6% of the host's RAM. Over 31 hours on a single NVIDIA H200 accelerator (with 16 CPU cores and 200 GB of system memory), disk usage grew steadily from 28 GB to 76 GB as checkpoints and logs accumulated. GPU utilization stabilized above 90% shortly after the warmup phase and remained near saturation for the remainder of training. To sum up, our setup runs comfortably on a single GPU node with modest additional CPU and memory overhead.

Embedder model evaluation Experiments were conducted on a single MacBook Pro with an M1 chip. The only computationally expensive part of the experiments (for the local machine) is launching multiple Coq-LSP servers at once (CoqPilot benchmark does that to optimize the time of the experiments and accelerate type-checking). As we use a middleware service over LLM APIs, our financial estimations might not be accurate. However, we roughly estimate 12 generation attempts per theorem with seven contextual theorems at 12 cents per theorem for Claude 3.5 and 7 cents for GPT-4o. Running the experiment on 300 theorems and repeating it three times amounts to a total of 171 US Dollars. Performing the same experiment for three different retrieval engines brings the overall cost to approximately 513 US Dollars.

Agent evaluation In the case of the agent evaluation, we ran the experiment only once and did not provide the confidence intervals due to financial limitations. We ran our agent on the IMM-300 dataset, and afterward, we compared five versions of the agent on the IMM-50 dataset in our § 4.3. That results in 550 attempts to prove different theorems. We estimate a single attempt at 1.3 US dollars. Therefore, we estimate the cost of the evaluation of the agent to be 715 US dollars.